Semantic Document Search Engine Documentation

Table of Contents

Overview

The Semantic Document Search Engine is a desktop application that leverages natural language processing and vector embeddings to understand and answer questions about uploaded documents. Unlike traditional keyword-based search systems, this application understands the semantic meaning behind both documents and user queries, allowing for more intelligent and context-aware responses.

The application supports PDF and TXT files, which are processed into embeddings and stored in a vector database. When users ask questions, the system retrieves the most relevant document sections and uses a language model to generate comprehensive answers based on those sections.

System Architecture

The application follows a modular architecture with these main components:

1. **User Interface (UI)** - A PyQt6-based interface that allows users to:
   - Upload PDF and TXT documents
   - Ask questions about the documents
   - View AI-generated answers

2. **Document Processing Pipeline**:
    - Text extraction from PDFs and TXT files
    - Text chunking for better context management
    - Vector embedding generation via OpenAI's text-embedding-3-large model
3. **Vector Database**:
    - SQLite-based persistent storage
    - Vector similarity search functionality
    - Document reference tracking
4. **Question Answering System**:
    - Context retrieval based on vector similarity
    - LLM-powered answer generation
    - Conversation history management

Installation and Setup

Prerequisites

- Python 3.6+
- OpenAI API key
- PyQt6
- Required Python packages: openai, numpy, PyPDF2, sklearn, dotenv

Environment Setup

1. Clone the repository
2. Create a

    ```
    .env
    ```

    file in the project root with:

    ```
    OPENAI_API_KEY=your_openai_api_key
    OPEN_AI_MODEL=gpt-3.5-turbo  # or your preferred OpenAI model
    ```

Installation

```
# Install dependencies
pip install -r requirements.txt
```

Usage Guide

Starting the Application

Run the application using:

```
python -m Midterm.main
```

Uploading Documents

1. Click the "Upload Document" button
2. Select one or more PDF or TXT files from the file dialog
3. Wait for the confirmation message that documents have been processed

Asking Questions

1. Type your question in the text field
2. Press Enter or click the "Ask" button
3. View the generated answer in the answer area

Best Practices

- Upload documents relevant to your questions
- Ask specific questions for more accurate answers
- For complex topics, upload multiple documents that cover different aspects

Technical Details

Document Processing Flow

1. **Text Extraction**:
   - PDF files: Uses PyPDF2 to extract text from each page
   - TXT files: Directly reads content
2. **Text Chunking**:
   - Text is divided into smaller chunks of approximately 500 characters
   - 50-character overlap between chunks maintains context across boundaries
   - `split_text_numpy` function handles efficient chunking
3. **Embedding Generation**:
   - Each text chunk is sent to OpenAI's embedding service
   - The "text-embedding-3-large" model converts text into 1536-dimensional vectors
   - Embeddings capture semantic meaning of text chunks
4. **Database Storage**:
   - Embeddings stored as binary blobs in SQLite
   - Additional metadata stored: filename, text content, creation timestamp

Question Answering Process

1. **Query Embedding**:
   - User question is converted to embedding vector using same model
2. **Similarity Search**:
   - Cosine similarity computed between question and stored embeddings
   - Top k (default: 3) most similar chunks retrieved
3. **Context Assembly**:
   - Retrieved chunks formatted with source information
   - Assembled into coherent context for the LLM
4. **Answer Generation**:
   - Context and question sent to OpenAI's chat completion API

- System prompt guides response format
- Response displayed to user with source attribution

Design Choices

Vector Database Implementation

The application uses a custom `VectorDB` class built on top of SQLite rather than specialized vector databases for several reasons:

- **Portability**: SQLite requires no separate server process, making the application self-contained
- **Simplicity**: Reduces dependencies and complexity
- **Persistence**: Data persists between application sessions
- **Custom Similarity**: Implements cosine similarity for semantic matching

Text Chunking Strategy

The chunking approach (500 chars with 50 char overlap) balances:

- **Context preservation**: Enough text to maintain meaning
- **Specificity**: Small enough to target relevant information
- **Efficiency**: Manageable embedding costs and storage requirements

UI Design Choices

The dark-themed UI follows modern design principles:

- **Minimal and focused**: Limited to essential functions
- **Intuitive workflow**: Clear progression from upload to question to answer
- **Visual feedback**: Status indicators during processing
- **Responsive layout**: Adapts to different screen sizes

OpenAI Integration

The application is designed to work with OpenAI models, with:

- **Model flexibility**: Configurable through environment variables

- **Temperature setting**: Lower temperature (0.3) for more focused answers
- **Token management**: Limited to 500 output tokens for concise responses
- **System prompt customization**: Guides the model to answer from documents and cite sources